Verbesserung der Leistungsfähigkeit von Caches

Peter Marwedel
Informatik 12
TU Dortmund

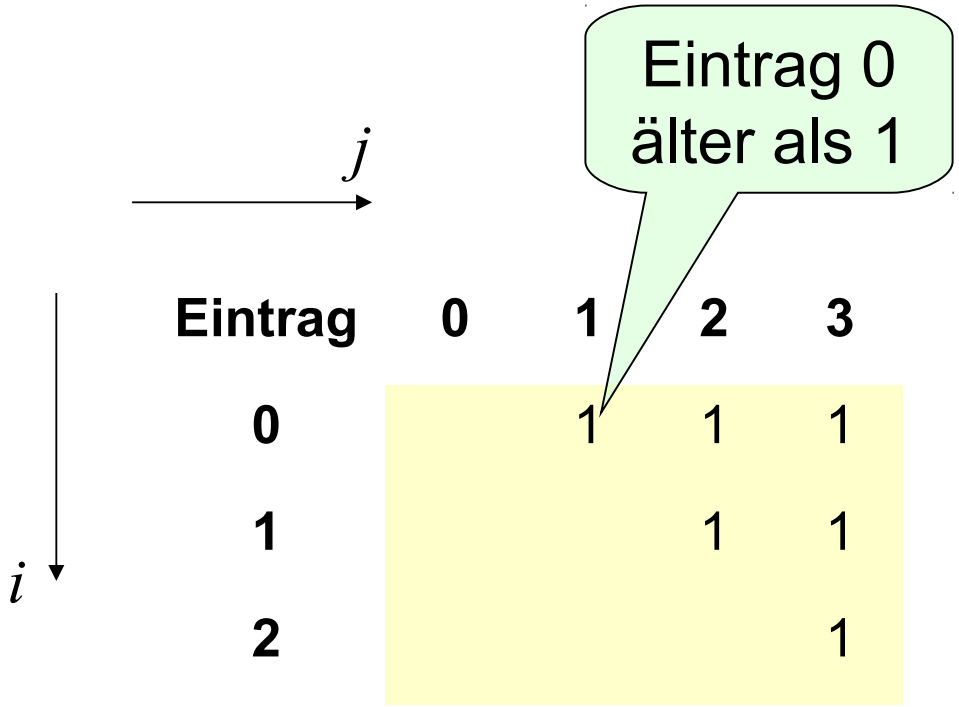
2011/05/20

Vier Fragen für Caches

1. Wo kann ein Speicherblock im Cache abgelegt werden (*block placement*)
2. Wie wird ein Speicherblock gefunden (*block identification*)
3. Welcher Block sollte bei einem Fehlzugriff (*cache miss*) ersetzt werden? (*block replacement*)
4. Was passiert beim Schreiben von Daten in den Cache? (*write strategy*)

Realisierung von LRU mittels Dreiecksmatrix (1)

Def: $f[i, j] = 1$, falls Zugriff auf i älter ist als der auf j und
0 sonst.



Wegen Antisymmetrie nur Dreiecksmatrix $f[i, j]$, $i < j$ (d.h. Zeilenindex < Spaltenindex) zu speichern.

Realisierung von LRU mittels Dreiecksmatrix (2)

Operationen:

1. Zugriff auf k: k wird jüngster Eintrag:
   $\forall\, j > k : f[k, j] = 0; \forall i < k : f[i, k] = 1$
2. *Miss* im *Cache*: gesucht wird ältester Eintrag, d.h. Eintrag k, für den gilt: $\forall\, j > k : f[k, j] = 1 \wedge \forall\, i < k : f[i, k] = 0$

Annahme: Initialisierung mit 1

Cache miss ☝ Überschreiben von *way* 0 →

Cache miss ☝ Überschreiben von *way* 1 →

Jüngster Eintrag jeweils gestrichelt

Eintrag	0	1	2	3
0		1	1	1
1			1	1
2				1

Eintrag	0	1	2	3
0		0	0	0
1			1	1
2				1

Eintrag	0	1	2	3
0		1	0	0
1			0	0
2				1

Realisierung von LRU mittels Dreiecksmatrix (3)

Cache miss ⇒ Überschreiben von *way* 2

Ein-trag	0	1	2	3
0		1	0	0
1			0	0
2				1

→

Ein-trag	0	1	2	3
0		1	1	0
1			1	0
2				0

Cache miss ⇒ Überschreiben von *way* 3

→

Ein-trag	0	1	2	3
0		1	1	1
1			1	1
2				1

Cache miss ⇒ Überschreiben von *way* 0

→

Ein-trag	0	1	2	3
0		0	0	0
1			1	1
2				1

Hit on way 2 ⇒ way 2 wird jüngster →

Ein-trag	0	1	2	3
0		0	1	0
1			1	1
2				0

Realisierung von LRU mittels Dreiecksmatrix (4)

Hit on way 0 ⇒ *way* 0 wird jüngster →

Ein-trag	**0**	**1**	**2**	**3**
0		0	1	0
1			1	1
2				0

Hit on way 3 ⇒ *way* 3 wird jüngster →

Ein-trag	**0**	**1**	**2**	**3**
0	0	0	0	
1		1	1	
2			0	

Ein-trag	**0**	**1**	**2**	**3**
0		0	0	1
1			1	1
2				1

Write allocate vs. *No-write allocate*

2 Varianten hinsichtlich des Schreibens:

Write allocate

- Block wird bei fehlgeschlagenem Schreibzugriff im Cache alloziert (entspricht Verhalten bei *read miss*)

No-write allocate

- Block bei *write miss* nur auf nächster Hierarchiestufe aktualisiert

Beliebig kombinierbar mit *write back/through*, i.d.R. aber:

- *Write back* + *write allocate*
- *Write through* + *no-write allocate*

Verbesserung der Leistungsfähigkeit von Caches ($, €, ¥): Übersicht

- Beeinflussende Größen (Kapitel 5.2, Hennessy, 3./4. Aufl.):
  - *Hit Time*
  - *Cache Bandwidth*
  - *Miss Penalty*
  - *Miss Rate*

Wir betrachten daher im folgenden Verfahren zur:

- Reduktion der Zugriffszeit im Erfolgsfall,
- Erhöhung der Cache-Bandbreite
- Reduktion der Wartezyklen, die ein fehlgeschlagener Zugriff erzeugt,
- Reduktion der Rate fehlgeschlagener *Cache*-Zugriffe.

Hit Time ↓: Kleine und einfache Caches

Zugriffszeit im Erfolgsfall (*hit time*) kritisch für Leistung von Caches, da sie CPU-Takt begrenzt

- Indizierung des *Tag*-Speichers (via Index) und Vergleich mit angefragter Adresse ☝ schnell, wenn *Cache* klein
- Kleiner Cache kann *on-chip* (der CPU) realisiert werden
  ⇒ kürzere Signallaufzeiten, effizienterer Datentransfer
- Einfacher, d.h. *direct mapped*:
  ⇒ Überlappung von Datentransfer und *Tag*-Prüfung möglich

⇒ Größe des L1-*Caches* ggf. nicht erhöht, evtl. sogar reduziert (Pentium III: 16KB, Pentium 4: 8KB)

Schwerpunkt auf höherer Taktrate, Kombination mit L2 *Cache*

Hit Time ↓: Kleine und einfache Caches

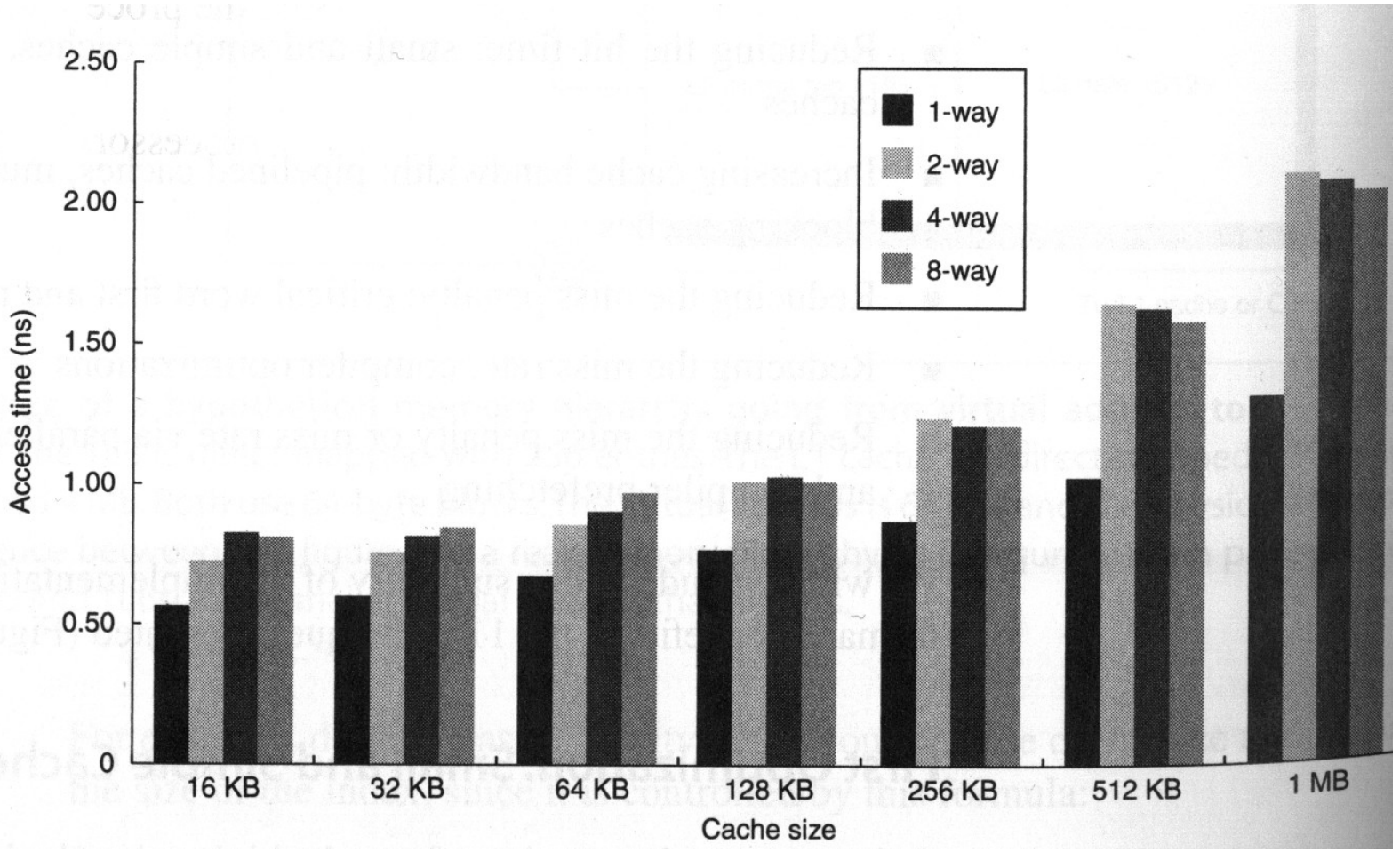


Abb. 5.4. aus HP07, © Elsevier

Hit Time ↓: Way Prediction

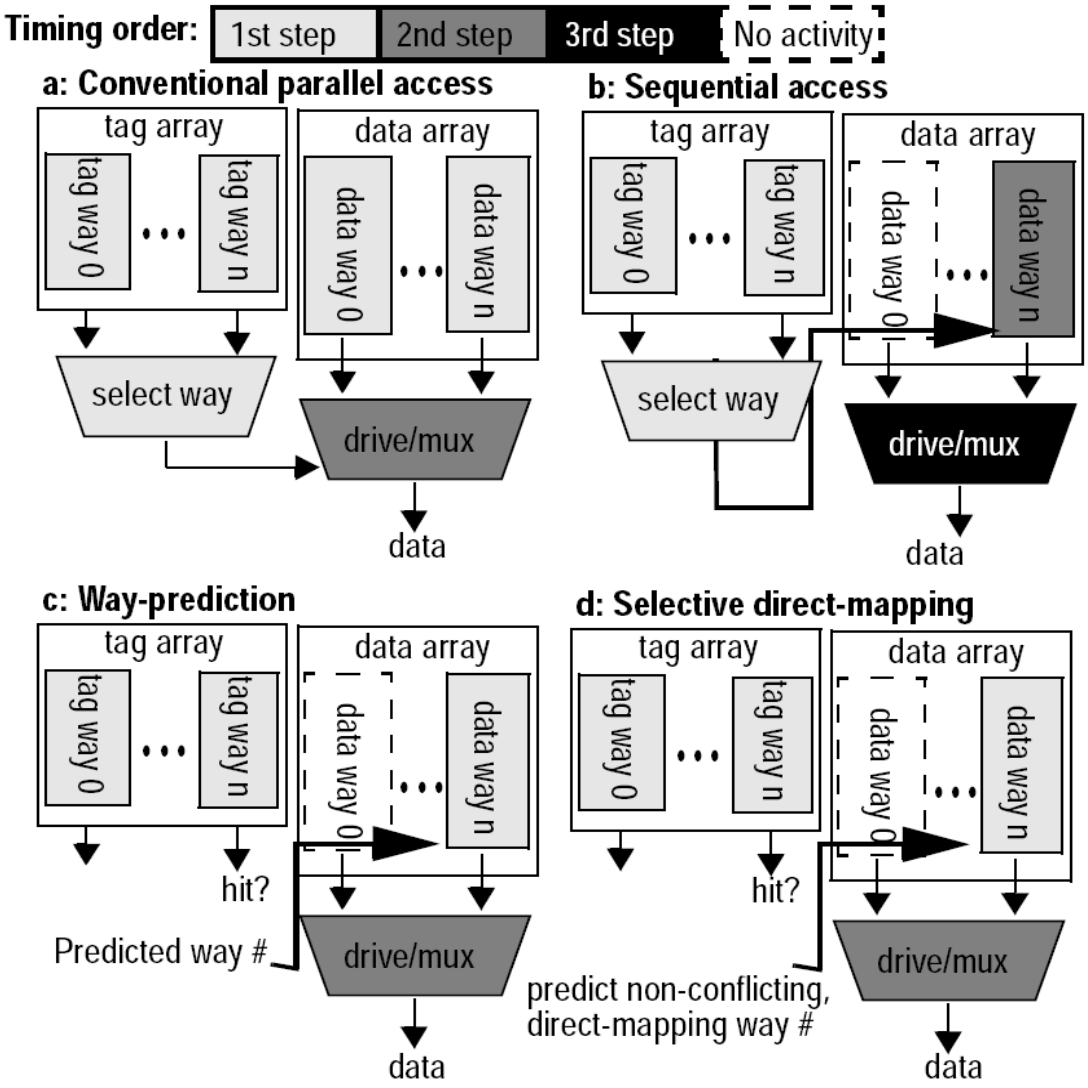
Idee: Zugriffsgeschwindigkeit eines *direct mapped* Caches erhalten und trotzdem Konflikte reduzieren

Methode: Zusätzliche Bits im *Cache*-Set zur Vorhersage des nächsten Zugriffs (in diesem Set)

- Multiplexer zum Auslesen der Daten kann früh gesetzt werden
- Nur ein *Tag*-Vergleich erforderlich (im Erfolgsfall)
- Weitere Tags nur geprüft bei (potentiellem!) *cache miss* (erfordert dann weitere Taktzyklen)

Way-prediction macht Caches energieeffizienter!

Hit Time ↓: Way prediction (2)



[M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, K. Roy: Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping, *MICRO-34*, 2001]

Beispiel: Alpha 21264:
1 Bit für *way prediction* in 2-Wege assoziativem (Befehls-) Cache
 Latenz 1 Takt bei korrekter (85% für SPEC95) und
3 Takte bei falscher Vorhersage

Hit Time ↓: Trace Caches

Prinzip:

- Befehle werden nur entlang möglicher Ausführungsfolgen gespeichert; *Cache* bestimmt dynamische Befehlssequenz, die in *Cache* geladen wird
- Einträge beginnen an beliebiger Adresse
- Blöcke gehen über Sprungbefehle hinaus

 Sprungvorhersage wird Teil des *Cache*-Systems!

- *Traces* werden ggf. mehrfach in *Cache* geladen als Folge von bedingten Sprungbefehlen mit variierendem Verhalten

Beispiel: Intel NetBurst Mikroarchitektur (Pentium 4)

Hit Time ↓: Keine Adressumsetzung bei Cache-Zugriff

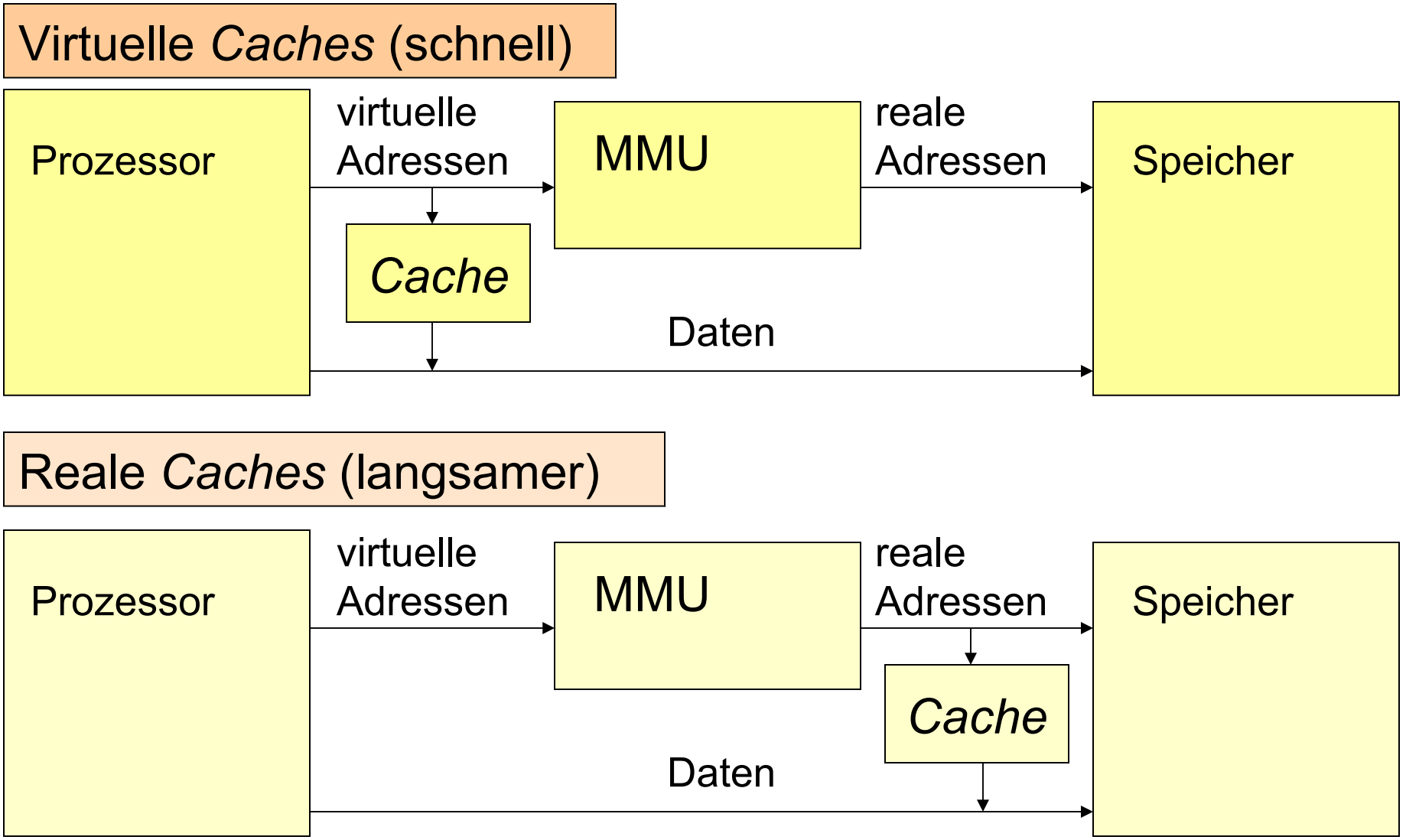
Aktuelle PCs verwenden virtuellen Speicher:

Virtuelle oder reales Caches?

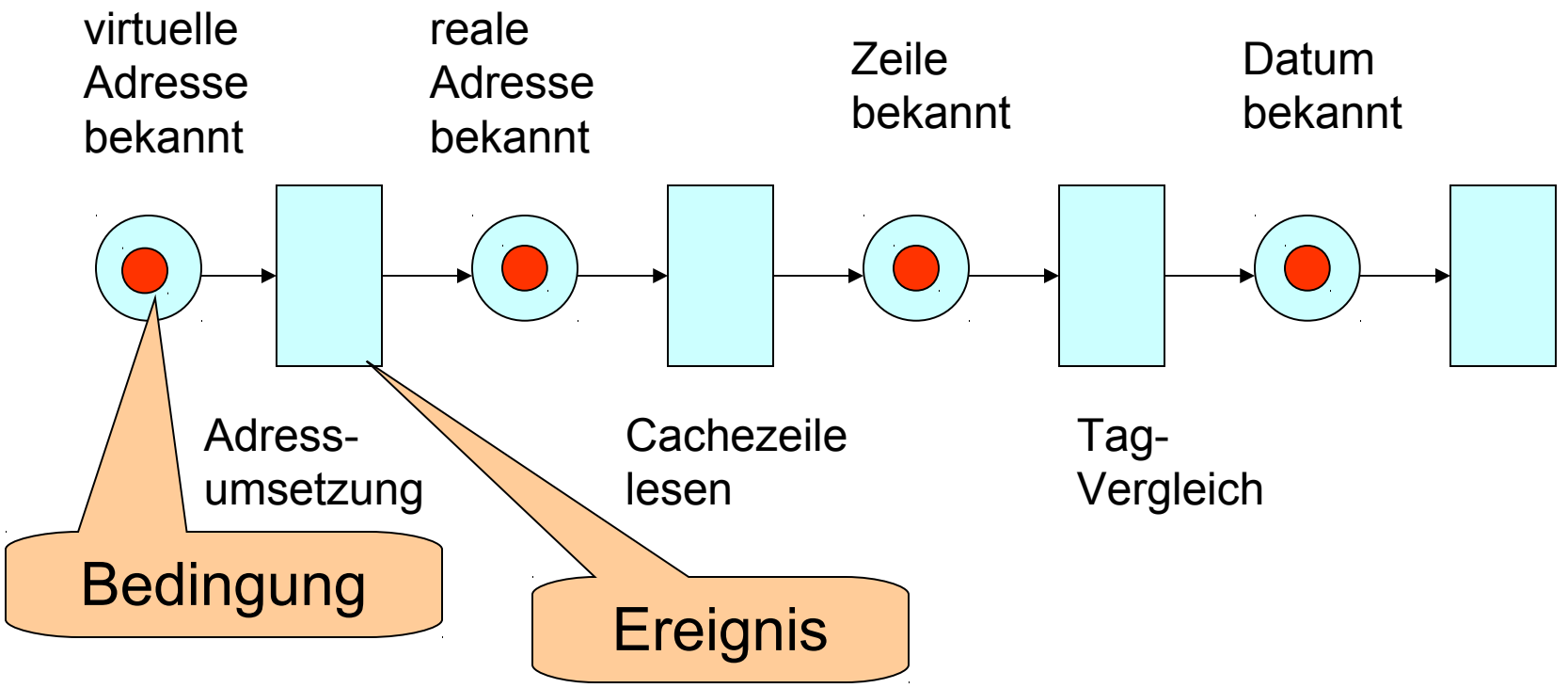
Siehe Kurs „Rechnerstrukturen“

Hier: Zusätzliche Betrachtung von Mischformen

Zur Erinnerung aus RS

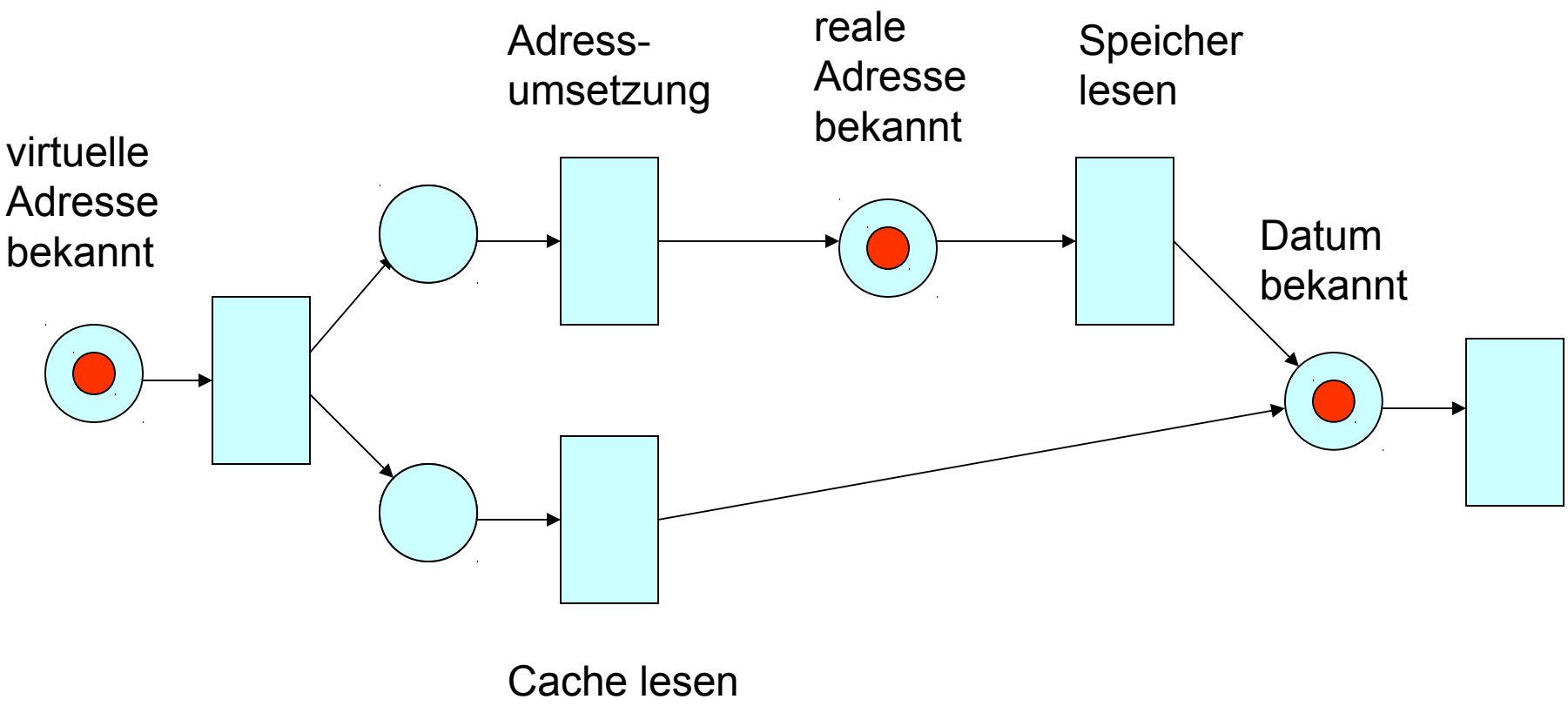


Modell des Ablaufs bei realen Caches



Adressumsetzung und Lesen des Caches nacheinander

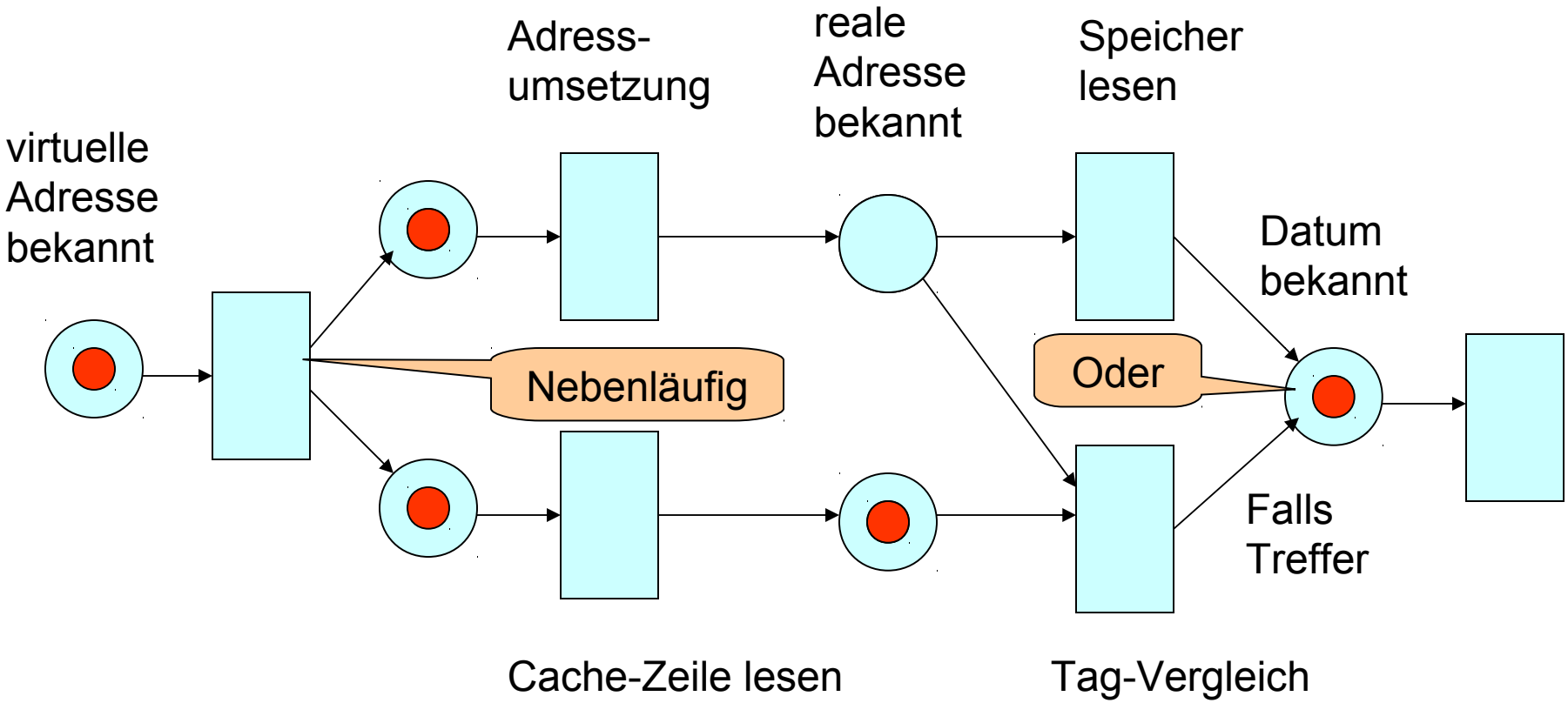
Modell des Ablaufs bei virtuellen Caches



Lesen des Caches und Adressumsetzung „parallel“

Mischform „*virtually indexed, real tagged*“

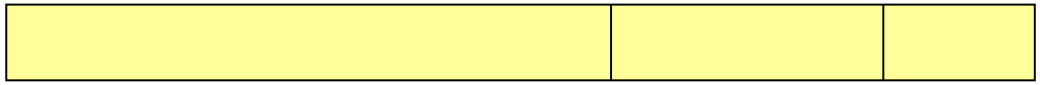
Indexwerte für virtuelle und reale Adressen identisch
 Zugriff auf realen Cache kann bereits während der Umrechung auf reale Adressen erfolgen.



Mischform „*virtually indexed, real tagged*“ (2)

Nachteil:

- Sei eine „*Meta-Page*“ der Speicherbereich, der zu einem bestimmten Tag-Feld gehört (unsere Bezeichnung)
- Innerhalb jeder „*Meta-Page*“ müssen virtueller und realer Index übereinstimmen.
- De facto bedeutet das, dass wir statt Seiten nunmehr „*Meta-Pages*“ als Einheiten der Speicherzuteilung nutzen. Damit wird aber die Speicherzuteilung sehr grob-granular.
- Beispiel: *direct mapped* $ mit 256 kB = 2^{18} B, mit 64 Bytes pro $-Eintrag 12 Bits für den Index, Meta-Pages von 256 kB.



Unterschiedliche Kombinationsmöglichkeiten

Index/ Tag	**Virtuell**	**Real**
Virtuell	*Virtually indexed, virtually tagged* (VIVT) Schnell, Kohärenz-probleme	*Physically (real)-indexed, virtually tagged* Theoretisch möglich, aber sinnlos (Geschwin-digkeit wie PIPT)
Real	*Virtually indexed/ physically (real)- tagged* (VIPT) Mittelschnell	*Physically (real)-indexed*, *physically (real)-tagged* (PIPT) Langsam, keine Kohärenzprobleme

Miss Penalty ↓: Nonblocking Caches

Prinzip: Überlappen der Ausführung von Befehlen mit Operationen auf der Speicherhierarchie

Betrachten CPUs mit *out-of-order completion/execution*
 Anhalten der CPU bei Fehlzugriff auf Cache nicht zwingend erforderlich (z.B. weiteren Befehl holen, während Wartezeit auf Operanden)

Notwendig: *Cache* kann trotz vorangegangen (und ausstehenden) Fehlzugriffs nachfolgende erfolgreiche Zugriffe (*hits*) befriedigen
 bezeichnet als "*hit under miss*"
Auch: "*hit under multiple misses*" bzw. "*miss under miss*"

Cache Bandwidth ↑: Multibanked Caches

Blöcke werden über mehrere Speicherbänke verteilt

Block-adresse	Bank 0
0	
4	
8	
12	

Block-adresse	Bank 1
1	
5	
9	
13	

Block-adresse	Bank 2
2	
6	
10	
14	

Block-adresse	Bank 3
3	
7	
9	
15	

Cache Bandwidth ↑: Pipelined Cache Access

Cache Zugriff über mehrere Stufen der Pipeline realisieren/verteilen

 Latenzzeiten des L1-Caches können größer sein als bei Zugriff in einem Takt (Pentium 4: 4 Zyklen)

Konsequenz mehrerer Pipelinestufen

- Größere Verzögerung für Sprungbefehle (*branch penalty*) bei falscher Sprungvorhersage
- Längere Verzögerung zwischen Laden und Verwenden von Operanden (*load delay*) siehe MIPS R4000!

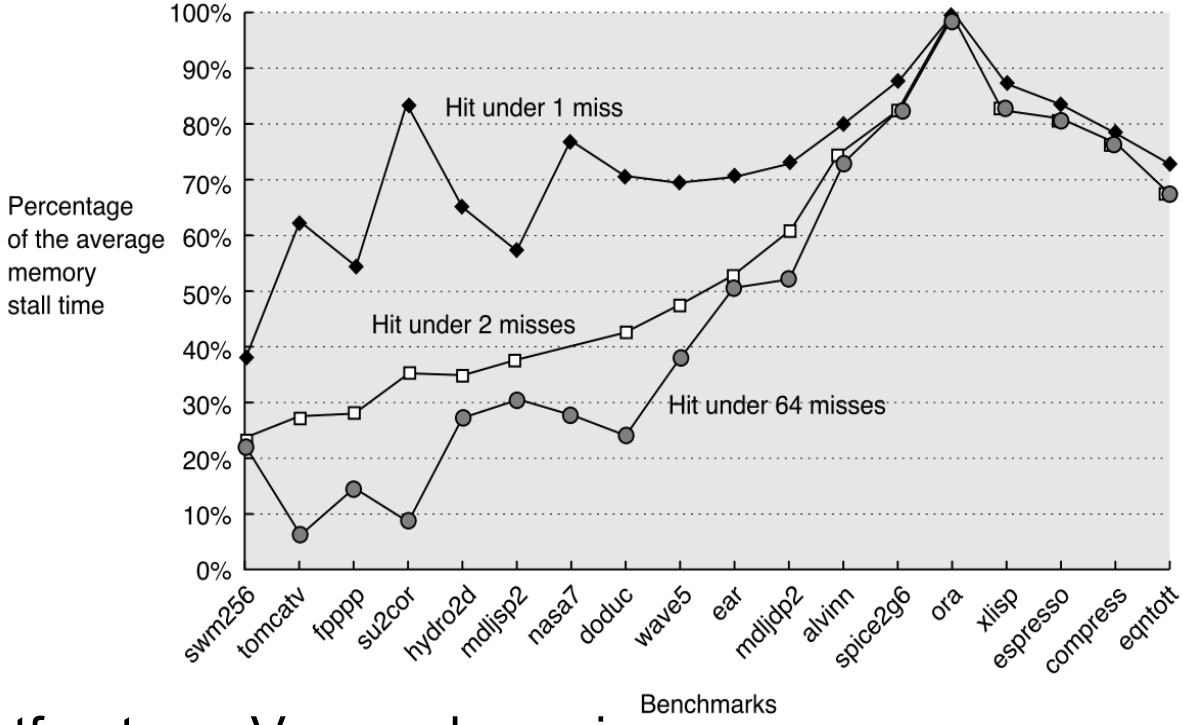
Beachte:

- *Cache*-Latenz ändert sich nicht, aber sie wird versteckt.

Miss Penalty ↓: Nonblocking Caches (2)

Nicht-blockierende Caches „*hit under (multiple) misses*”:
Vorteil gegen blockierenden Cache (SPEC92)

- *hit under 64 misses* = 1 miss pro 64 Registern
- Verbesserung durch *hit under multiple misses* haupsächlich bei FP-Benchmarks



Verwendeter Compiler plant Ladeoperationen entfernt von Verwendung ein.
Jetzt mehrere ausstehende Zugriffe auf den gleichen Cache-Block möglich ☝ muss geprüft werden

Miss Penalty ↓: Critical Word First & Early Restart

CPU benötigt häufig nur einen Wert aus dem *Cache*-Block

 nicht warten, bis kompletter Block nachgeladen

2 Realisierungsmöglichkeiten:

- *Critical Word First*
  - Zuerst angefragtes Datenwort aus Speicher holen
  - CPU-Ausführung sofort fortsetzen lassen
  - Weitere Inhalte dann parallel zur CPU holen
- *Early Restart*
  - Daten in normaler Reihenfolge in Cache-Block laden
  - CPU-Ausführung fortsetzen, sobald angefragtes Wort geladen

Vorteile nur bei großen Cache-Blöcken

Mit örtlicher Lokalität Wahrscheinlichkeit groß für Zugriff im aktuellen Block!

Miss Penalty ↓: Zusammenfassung von Schreibpuffern

Betrachte *write through Cache*

Schreibzugriffe in (kleinem) Puffer gespeichert und bei Verfügbarkeit des Speicherinterfaces ausgeführt
Hier: Daten Wort-weise geschrieben!

Annahme: Schreiben größerer Datenblöcke effizienter möglich (trifft auf heutige Speicher in der Regel zu!)

 Schreibpuffer im *Cache* gruppieren, falls zusammenhängender Transferbereich entsteht

Miss Penalty ↓: Zusammenfassung von Schreibpuffern (2)

Write buffer merging:

- Oben: ohne
- Unten: mit

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Beachte: Beispiel extrem optimistisch!

Falls *write buffer merging* nicht vorgesehen, ist Schreibpuffer auch nicht mehrere Worte breit!

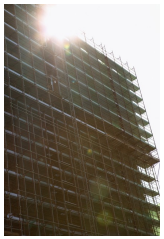
Miss Penalty ↓: Einführung mehrerer *Cache*-Ebenen

- Unabhängig von der CPU
- Fokus auf Schnittstelle *Cache*/Hauptspeicher

Dilemma: Leistungsfähigkeit des *Caches* steigern durch

- Beschleunigung des *Caches* (um mit CPU „mitzuhalten”) [*Cache* muss kleiner werden] oder
- Vergrößerung des *Caches* (um viele Anfragen an Speicher „befriedigen” zu können) [Cache langsamer]

 Eigentlich beide notwendig, aber unvereinbar!



Lösung: 2 *Cache*-Ebenen

- 1st-*level Cache* (L1): schnell, klein, angepasst an CPU-Takt
- 2nd-*level Cache* (L2): groß genug, viele Zugriffe zu puffern, kann langsamer sein

Miss Penalty ↓: Einführung mehrerer *Cache*-Ebenen (2)

Leistungsbewertung jetzt komplexer:

Interaktion von L1 und L2-Cache muss berücksichtigt werden!

(1) Mttl. Zugriffszeit = *Hit Time*(L1) + *Miss Rate*(L1) x *Miss Penalty* (L1)
mit Charakteristik des L2-Caches

(2) *Miss Penalty*(L1) = *Hit Time*(L2) + *Miss Rate*(L2) x *Miss Penalty*(L2)

(3) Mttl. Zugriffszeit = *Hit Time*(L1) + *Miss Rate*(L1) x
(*Hit Time*(L2) + *Miss Rate*(L2) x *Miss Penalty*(L2))

Beachte:

- *Miss Rate*(L2) gemessen auf von L1 „übrigen" Zugriffen
- Unterscheidung von lokaler und globaler Fehlzugriffsrate
(lokal groß für L2-*Cache*, da nur auf Fehlzugriffen von L1)

Miss Penalty ↓: Einführung mehrerer *Cache*-Ebenen (3)

Parameter von 2-Ebenen-Cache-Systemen:

- Geschwindigkeit von L1-Cache beeinflusst CPU-Takt
- Geschwindigkeit von L2-Cache beeinflusst nur Verzögerung bei Fehlzugriff auf Daten in L1-Cache => (relativ?) selten

☝Viele Alternativen für L2, die bei L1 nicht anwendbar sind

Ziel: L2-Cache (wesentlich) größer als L1-Cache!

Verhältnis von L1- zu L2-Cache-Inhalten:

- *Multilevel Inclusion* (Daten in L1 immer in L2 vorhanden)
  ☝Konsistenz(prüfung) einfach!
  Beachte: Setzt deutlich größeren L2-Cache voraus als s.u.
- *Multilevel exclusion* (Daten von L1 nie in L2 vorhanden)
  L1-Fehlzugriff bewirkt Vertauschung von Blöcken zw. L1 u. L2
  ☞ speicherökonomisch, falls L2 nur gering größer als L1
  (z.B. AMD Athlon: L1 = 64kB, L2 = 256kB)

Miss Penalty ↓: Priorisierung von Lese- über Schreibzugriffen

Idee: Lesezugriffe behandeln,
bevor Schreiboperation abgeschlossen

Problem bei *write through*:

Wichtigste Optimierung: Schreibpuffer
☝ kann Wert enthalten, der von Lesezugriff angefragt ist

Lösungsmöglichkeiten:

- Warten bis Schreibpuffer leer (kontraproduktiv)
- Inhalte des Schreibpuffers prüfen und mit Lesezugriff fortfahren, falls keine Konflikte (üblich in *Desktop/Server*)

Vorgehen bei *write back*:

- „*dirty*" Block in (kleinen) Puffer übernehmen
- Zuerst neue Daten lesen, dann „*dirty*" Block schreiben

Miss Penalty ↓: *Victim Caches*

Problem: Verwerfen von *Cache*-Inhalten ungeschickt, falls bald wieder entsprechender Zugriff erfolgt

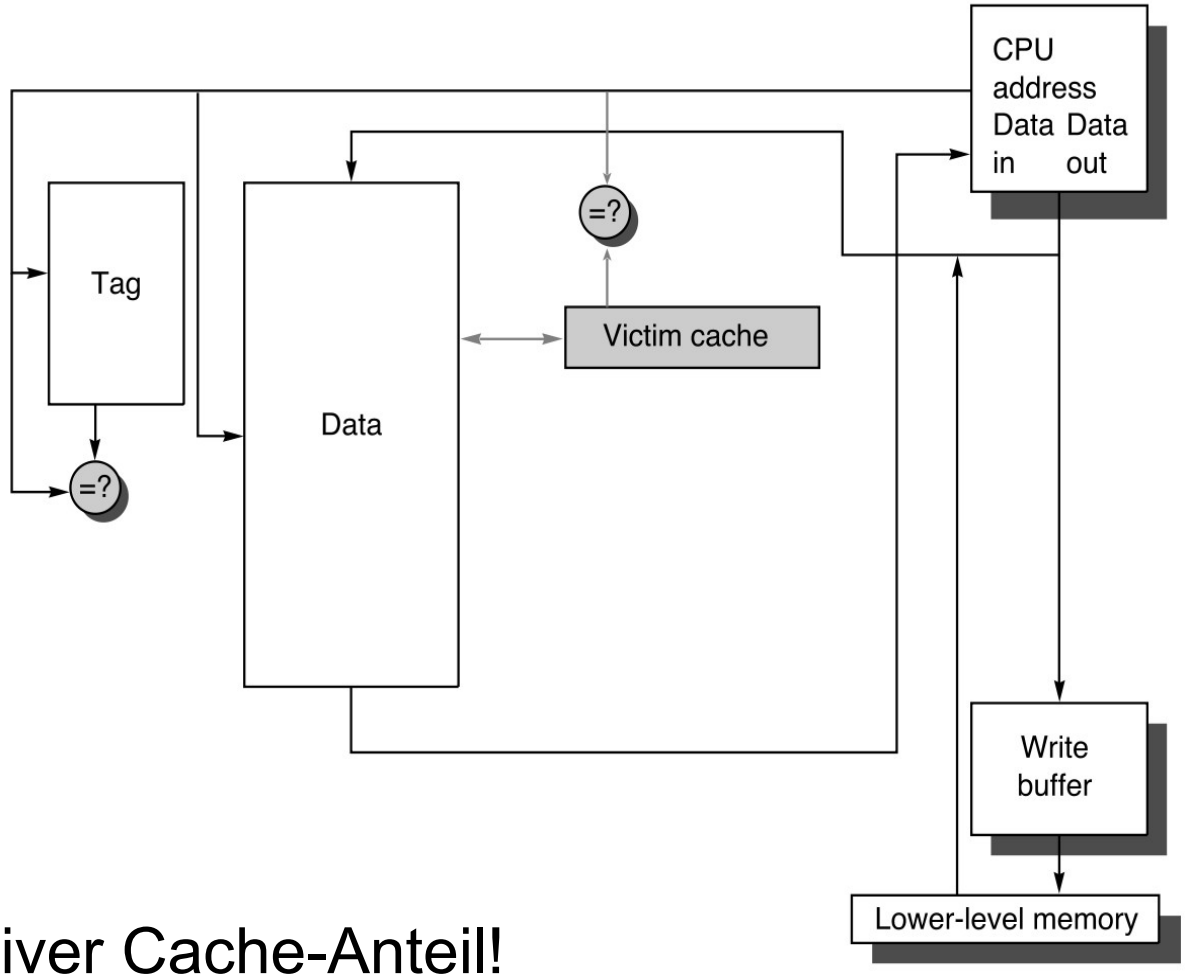
Idee: Verworfene Daten (kurz) puffern (= "Unter"-*Cache*)

- Erfordert kleinen, voll-assoziativen *Cache* zwischen eigentlichem Cache und der nächsten Hierarchieebene
- Bei Fehlzugriffen werden zuerst *victims* (= gerade verworfene Inhalte) geprüft, bevor Zugriff auf die nächste Hierarchieebene erfolgt

Verbesserung der Cache-Leistungsfähigkeit insbesondere bei kleinen, *direct mapped Caches* (bis zu 25% Zugriffe bei 4K-Cache)
AMD Athlon: *victim cache* mit 8 Einträgen: Ist das plausibel?
Was realisiert man mit dem "*Cache* im *Cache*"?

Miss Penalty ↓: *Victim Caches* (2)

Struktur eines *Caches* mit *Victim Cache*



 kleiner voll-assoziativer Cache-Anteil!

Miss Penalty ↓: *Prefetching* (durch HW oder SW)

(deutsch etwa „Vorabruf”, „vorher abholen”)

Idee: Elemente (Daten/Code) in *Cache* laden, bevor sie tatsächlich gebraucht werden

- *Prefetching* versucht Speicherbandbreite sinnvoll zu verwenden, die anderweitig ungenutzt bliebe
- Aber: Kann mit „regulären” Anfragen/Zugriffen interferieren!

a) Hardware-*Prefetching* für Instruktionen/Daten

- Bei *cache miss*: angefragter Block -> *Cache*, sequentieller Nachfolger -> “*stream buffer*”
- Falls bei späterer Anfrage Block im *stream buffer*: ursprüngliche Cache-Anfrage verwerfen, aus *stream buffer* nachladen und neuen *prefetch* starten

Miss Penalty ↓: *Prefetching* (durch HW oder SW) (2)

b) *Prefetching* kontrolliert durch Compiler

- Verwendet spezielle Befehle moderner Prozessoren
- Möglich für Register und *Cache*
- Beachte:
  - *Prefetch*-Instruktionen dürfen keine Ausnahmen (insbes. Seitenfehler) erzeugen *non-faulting instructions*
  - Cache darf nicht blockierend sein (damit Prefetch-Lade-vorgänge und weitere Arbeit der CPU parallel möglich)
  - *Prefetching* erzeugt *Overhead* Kompromiss erforderlich (nur für die Daten *prefetch* Instruktionen erzeugen, die mit großer Wahrscheinlichkeit Fehlzugriffe erzeugen)
- In Alpha 21264 auch *prefetch* für Schreibzugriffe (falls angeforderter Block komplett geschrieben wird: *write hint* Befehl alloziert diesen im *Cache*, aber Daten nicht gelesen!)

Miss Rate ↓
Compiler-Optimierungen

Fortschritt der Compiler-Technik ermöglicht Optimierungen, die Wissen über Speicherhierarchie einbeziehen

Verschiedene Bereiche unterscheidbar:

- Örtliche und zeitliche Lokalität der Zugriffe verbessern!
- Verbesserung des Zugriffs auf Code
  - Umordnung (z.B. Von Prozeduren) zur Reduktion von Fehlzugriffen durch Konflikte
  - Ausrichtung von Basisblöcken an Cache-Block-Grenzen
    kaum Fehlzugriffe mehr auf weiteren seq. Code
- Zahlreiche Verfahren Bestandteil der ES- und SES-Kurse

Miss Rate ↓
Compiler Optimierungen (2)

Umordnung von Schleifen

Feld (x[.][.]) zeilenweise gespeichert (*row major order*)

Problem: Zugriffe erfolgen nicht sequentiell
(Hier: verschachtelte Schleife Zugriff mit großer Schrittweite [100])

```
/* Vorher */
for (j = 0; j < 100; j++)
       for (i = 0; i < 5000; i++)
           x[i][j] = 2 * x[i][j]

/* Nachher */
for (i = 0; i < 5000; i++)
       for (j = 0; j < 100; j++)
           x[i][j] = 2 * x[i][j]
```

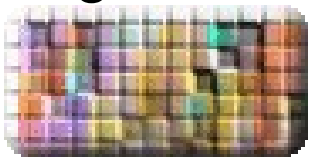
Falls Array nicht komplett in Cache viele Fehlzugriffe
(extrem, falls Cache-Größe gleich Zeilengröße des Arrays!)

Modifizierter Code greift sequentiell zu (sofern *row major*!)
 Effizienzverbesserung ohne Änderung der Anzahl ausgeführter Befehle;

Miss Rate ↓
Compiler Optimierungen (3)

Blockweise Bearbeitung

(*tiling*)



Idee: Fehlzugriffe reduzieren durch verbesserte temporale Lokalität

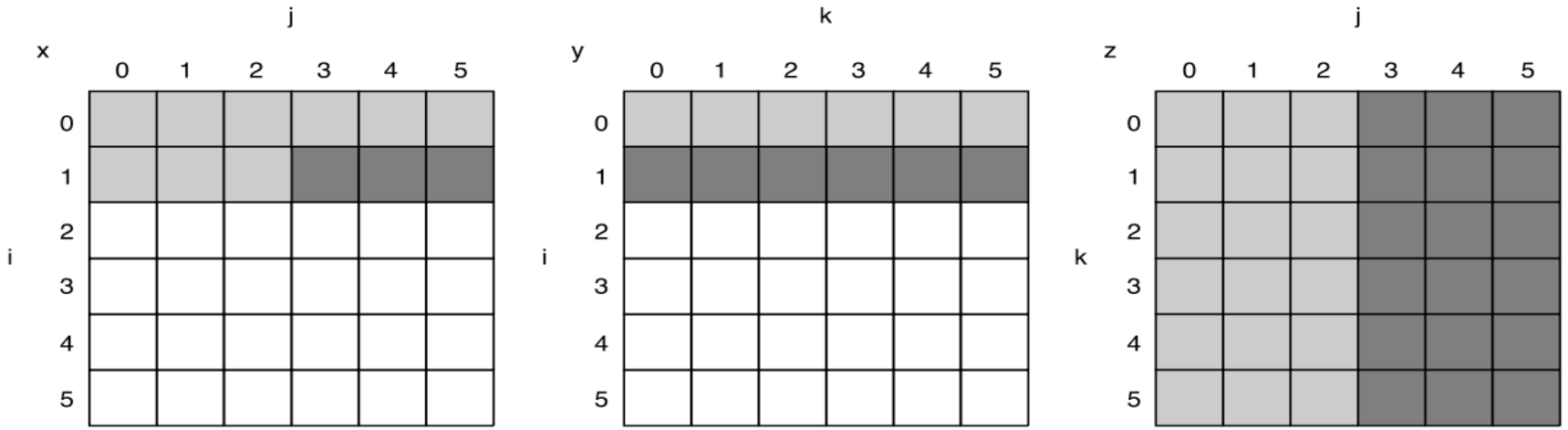
```
/* Vorher */
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
            r = 0;
            for (k = 0; k < N; k++)
                r = r + y[i][k]*z[k][j];
            x[i][j] = r;
            }
```

Betrachte Matrixmultiplikation

Problem: Zugriff sowohl zeilen- als auch spaltenweise
 Umordnung von Schleifen nicht hilfreich

Miss Rate ↓: Compiler Optimierungen (4)

Zugriffschema bei „naiver” Matrixmultiplikation
(„Alter” des Zugriffs über Einfärbung: dunkel = aktuell, mittel = länger zurückliegend, weiß = nicht angefragt)



 erfordert $2N^3 + N^2$ Datenzugriffe

Lösungsprinzip: Bearbeitung kleiner Teilblöcke der Daten (hier: Teilmatrizen)

Miss Rate ↓: Compiler Optimierungen (5)

Matrixmultiplikation mit „*blocking factor*" B:

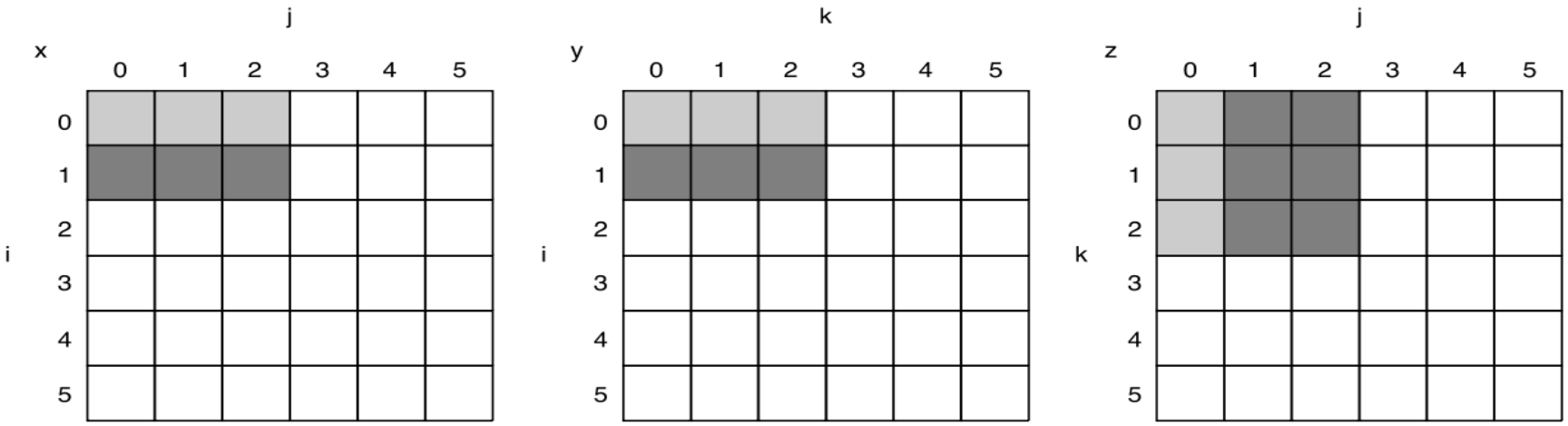
```
/* Nachher */
for (jj = 0; jj < N; jj += B)
for (kk = 0; kk < N; kk += B)
for (i = 0; i < N; i++)
   for (j = jj; j < min(jj+B,N); j++) {
         r = 0;
         for (k = kk; k < min(jj+B,N); k++)
            r = r + y[i][k]*z[k][j];
         x[i][j] = x[i][j] + r;
         }
```

Annahme: x[.][.] initial == 0, Block $B \times B$ passt in Cache

⇒ Erfordert $2N^3/B + N^2$ Zugriffe ☝ Verbesserung ca. Faktor B

Miss Rate ↓: Compiler Optimierungen (6)

Zugriffschema bei blockweiser Matrixmultiplikation:



 Nützt Kombination von temporaler Lokalität (z[.][.]) und örtlicher Lokalität (y[.][.]) aus

Extremfall: Kleine Blockgröße Daten können in Registern gehalten werden

Miss Rate ↓:

Klassische Methode zur Cache-Leistungssteigerung.

Typen von Cache-Misses:

- *Compulsory*

  Erster Zugriff auf Speicherbereich kann nie im Cache sein.

- *Capacity*

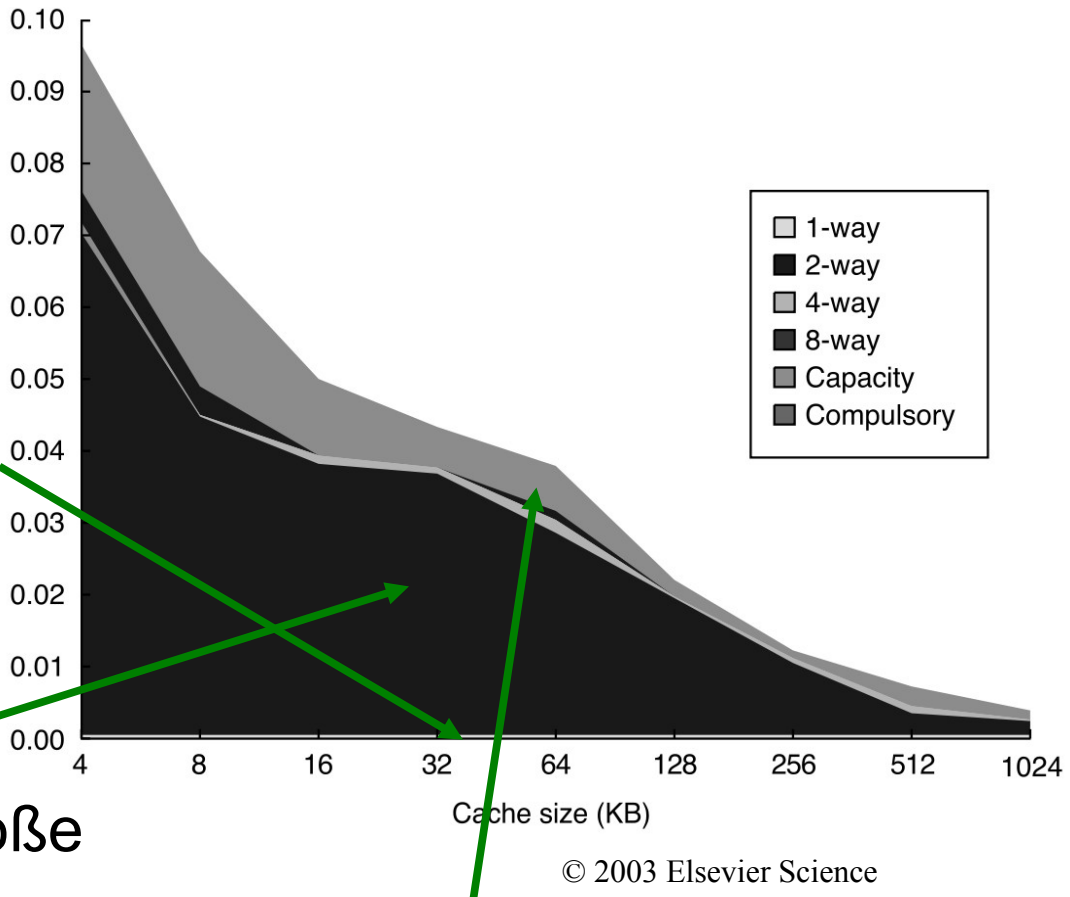
  Falls Cache nicht alle Blöcke enthalten kann, die für Programm-ausführung erforderlich Kapazitätsüberschreitung: Blöcke müssen verworfen und später wieder geholt werden!
  Worst Case: Cache viel zu klein Blöcke werden ständig ein-/ ausgelagert "*thrashing*"

- *Conflict*

  Bei *direct mapped* und n-Wege assoziativem $ können Konflikte bzgl. des Ablegens eines Blocks auftreten (d.h. zu viele Blöcke werden auf gleiches „Set" abgebildet)

Miss Rate ↓: (2)

Anteil der Fehlzugriffstypen (SPEC2000)



- Extrem wenige zwangsweise „*misses*" (normal bei lang laufenden Programmen)
- *Capacity misses* reduziert mit wachsender Cache-Größe
- Weitere Fehlzugriffe durch fehlende Assoziativität (Konflikte; bzgl. voll assoziativem *Cache*)

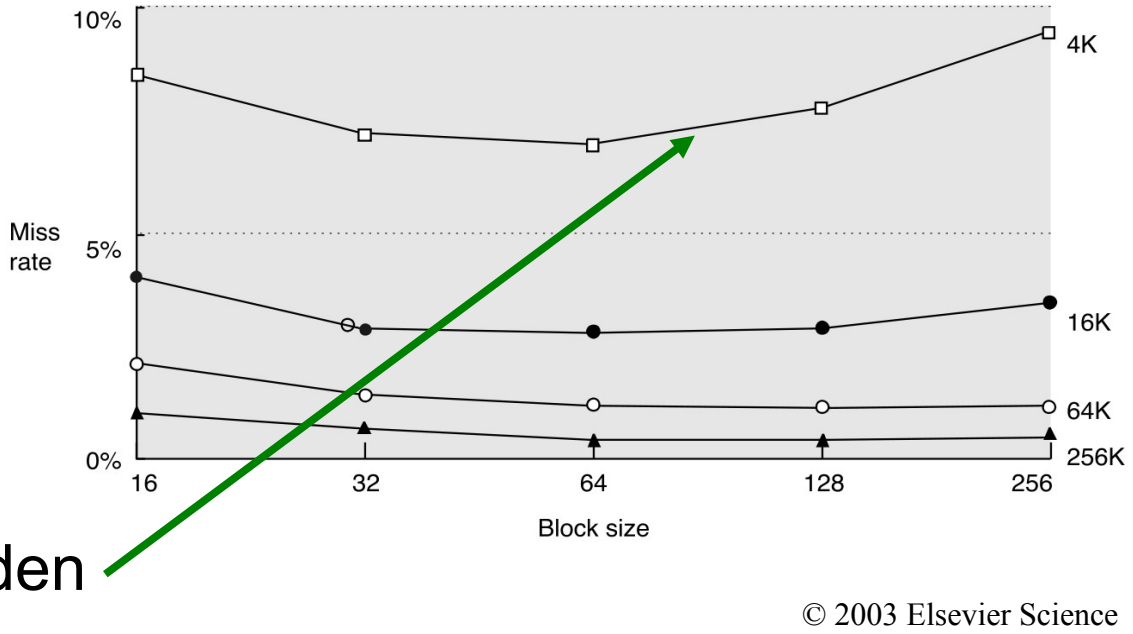
Miss Rate ↓: Größere Cache-Blöcke

Reduziert zwangsweise Fehlzugriffe wegen örtlicher Lokalität

Aber: *Miss penalty* wird erhöht

Auch: Da # Blöcke reduziert evtl. mehr Konflikt- und Kapazitätsfehlzugriffe

 Fehlzugriffsrate darf nicht erhöht werden



 Kosten des Fehlzugriffs dürfen Vorteile nicht überwiegen

- Abhängig von Latenz und Bandbreite des Speichers (hoch/hoch große Blöcke, niedrig/niedrig kleine)

Miss Rate ↓: Größere Caches

Nachteile:

- Längere Cache-Zugriffszeit
- Höhere Kosten

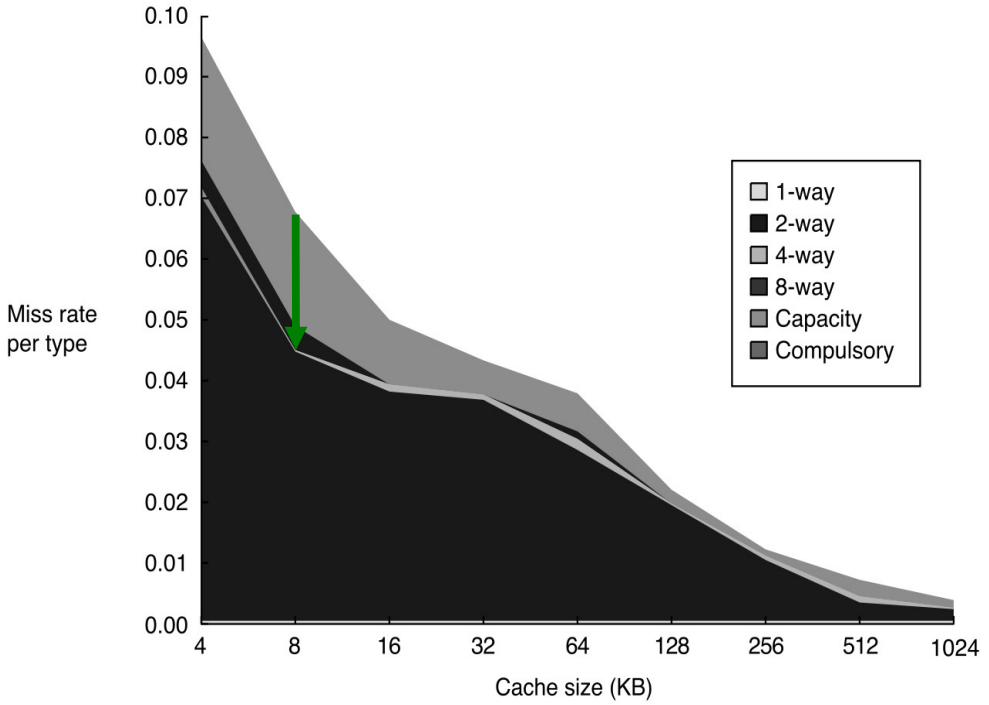
Bei 2-stufigen Caches ist Zugriffszeit der 2. Ebene nicht primär ausschlaggebend
 Hier größere Caches verwenden (ggf. langsamer)

Verbreitete Lösung: Große Caches als 2nd- oder 3rd-level Caches haben heute Kapazitäten wie Hauptspeicher früher!

Miss Rate ↓: Höhere Assoziativität

Assoziativität reduziert Fehlzugriffsrate

- In der Praxis 8-Wege assoziativ ≅ voll assoziativem Cache (aber: deutlich geringere Komplexität!)!



- „2:1-*Cache*-Daumenregel" *direct mapped* Cache der Größe N hat ca. Fehlzugriffsrate von 2-Wege assoziativem Cache halber Größe
- Hohe Assoziativität erhöht Zykluszeit, da Cache (1. Ebene) direkt mit CPU gekoppelt!

Zusammenfassung

Betrachten von Verfahren zur

- Reduktion der Wartezyklen, die ein fehlgeschlagener Zugriff erzeugt
- Reduktion der Rate fehlgeschlagener *Cache*-Zugriffe
- Reduktion der Gesamtkosten von Fehlzugriffen (*miss penalty* und *rate*) durch Parallelismus
- Reduktion der Zugriffszeit im Erfolgsfall
- Hauptspeicherorganisation